W3School Lua 教程

wizardforcel

Published with GitBook



目錄

W3School Lua 教程

作者：W3School

来源：Lua 教程

Lua 简介



Lua 是一种轻量小巧的脚本语言，用标准C语言编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Lua 是巴西里约热内卢天主教大学（Pontifical Catholic University of Rio de Janeiro）里的一个研究小组，由Roberto Ierusalimschy、Waldemar Celes 和 Luiz Henrique de Figueiredo所组成并于1993年开发。

设计目的

其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Lua 特性

- 轻量级: 它用标准C语言编写并以源代码形式开放，编译后仅仅一百余K，可以很方便的嵌入别的程序里。
- 可扩展: Lua提供了非常易于使用的扩展接口和机制：由宿主语言(通常是C或C++)提供这些功能，Lua可以使用它们，就像是本来就内置的功能一样。
- 其它特性:
  - 支持面向过程(procedure-oriented)编程和函数式编程(functional programming)；
  - 自动内存管理；只提供了一种通用类型的表（table），用它可以实现数组，哈希表，集合，对象；
  - 语言内置模式匹配；闭包(closure)；函数也可以看做一个值；提供多线程（协同进程，并非操作系统所支持的线程）支持；
  - 通过闭包和table可以很方便地支持面向对象编程所需要的一些关键机制，比如数据抽象，虚函数，继承和重载等。

Lua 应用场景

- 游戏开发

- 独立应用脚本
- Web 应用脚本
- 扩展和数据库插件如：MySQL Proxy 和 MySQL WorkBench
- 安全系统，如入侵检测系统

第一个 Lua 程序

接下来我们使用 Lua 来输出"Hello World!"

```
print("Hello World!")
```

运行后，会在屏幕上显示 Hello, world!。

Lua 环境安装

Linux 系统上安装

Linux & Mac上安装 Lua 安装非常简单，只需要下载源码包并在终端解压编译即可，本文使用了5.3.0版本进行安装：

```
curl -R -O http://www.lua.org/ftp/lua-5.3.0.tar.gz tar zxf lua-5.3.0.tar.gz
cd lua-5.3.0 make linux test
make install
```

Mac OS X 系统上安装

```
curl -R -O http://www.lua.org/ftp/lua-5.3.0.tar.gz tar zxf lua-5.3.0.tar.gz
cd lua-5.3.0 make macosx test
make install
```

接下来我们创建一个 helloWorld.lua:

```
print("Hello World!")
```

执行以下命令:

```
$ lua helloWorld
```

输出结果为：

```
Hello  World!
```

Window 系统上安装 Lua

window下你可以使用一个叫"SciTE"的IDE环境来执行lua程序，下载地址为：

- 本站下载地址：http://pan.baidu.com/s/1pJHqF8Z
- Github 下载地址：https://github.com/rjpcomputing/luaforwindows/releases
- Google Code下载地址：https://code.google.com/p/luaforwindows/downloads/list

双击安装后即可在该环境下编写 Lua 程序并运行。

你也可以使用 Lua 官方推荐的方法使用 LuaDist：http://luadist.org/ <split>123</split>

Lua 基本语法

Lua 学习起来非常简单，我们可以创建第一个 Lua 程序！

第一个 Lua 程序

交互式编程

Lua 提供了交互式编程模式。我们可以在命令行中输入程序并立即查看效果。

Lua 交互式编程模式可以通过命令 lua -i 或 lua 来启用：

```
$ lua -i
$ Lua 5.3.0  Copyright (C) 1994-2015 Lua.org, PUC-Rio
>
```

在命令行中，输入以下命令:

```
> print("Hello World！")
```

接着我们按下回车键，输出结果如下：

```
> print("Hello World！")
Hello World！
>
```

脚本式编程

我们可以将 Lua 程序代码保持到一个以 lua 结尾的文件，并执行，该模式称为脚本式编程，如我们将如下代码存储在名为 hello.lua 的脚本文件中：

```
print("Hello World！")
print("www.w3cschool.cc")
```

使用 lua 名执行以上脚本，输出结果为：

```
$ lua test.lua
Hello World！
www.w3cschool.cc
```

我们也可以将代码修改为如下形式来执行脚本（在开头添加：#!/usr/local/bin/lua）：

```
#!/usr/bin/lua

print("Hello World！")
print("www.w3cschool.cc")
```

以上代码中，我们指定了 Lua 的解释器 /usr/local/bin directory。加上 # 号标记解释器会忽略它。接下来我们为脚本添加可执行权限，并执行：

```
./test.lua
Hello World！
www.w3cschool.cc
```

注释

单行注释

两个减号是单行注释:

```
--
```

多行注释

```
--[[
 多行注释
 多行注释
 --]]
```

标示符

Lua 表示符用于定义一个变量，函数获取其他用户定义的项。标示符以一个字母 A 到 Z 或 a 到 z 或下划线 _ 开头后加上0个或多个字母，下划线，数字（0到9）。

最好不要使用下划线加大写字母的标示符，因为Lua的保留字也是这样的。

Lua 不允许使用特殊字符如 @, $, 和 % 来定义标示符。 Lua 是一个区分大小写的编程语言。因此在 Lua 中 W3c 与 w3c 是两个不同的标示符。以下列出了一些正确的标示符：

```
mohd         zara      abc     move_name    a_123
myname50     _temp     j       a23b9        retVal
```

关键词

以下列出了 Lua 的保留关键字。保留关键字不能作为常量或变量或其他用户自定义标示符：

and	break	do	else
elseif	end	false	for
function	if	in	local
nil	not	or	repeat
return	then	true	until
while			

一般约定，以下划线开头连接一串大写字母的名字（比如 _VERSION）被保留用于 Lua 内部全局变量。

全局变量

在默认情况下，变量总是认为是全局的。

全局变量不需要声明，给一个变量赋值后即创建了这个全局变量，访问一个没有初始化的全局变量也不会出错，只不过得到的结果是：nil。

```
> print(b)
nil
> b=10
> print(b)
10
>
```

如果你想删除一个全局变量，只需要将变量负值为nil。

```
b = nil
print(b)      --> nil
```

这样变量b就好像从没被使用过一样。换句话说，当且仅当一个变量不等于nil时，这个变量即存在。

Lua 数据类型

Lua是动态类型语言，变量不要类型定义,只需要为变量赋值。 值可以存储在变量中，作为参数传递或结果返回。

Lua中有8个基本类型分别为：nil、boolean、number、string、userdata、function、thread和table。

数据类型	描述
nil	这个最简单，只有值nil属于该类，表示一个无效值（在条件表达式中相当于false）。
boolean	包含两个值：false和true。
number	表示双精度类型的实浮点数
string	字符串由一对双引号或单引号来表示
function	由 C 或 Lua 编写的函数
userdata	表示任意存储在变量中的C数据结构
thread	表示执行的独立线路，用于执行协同程序
table	Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字或者是字符串。在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表。

我们可以使用type函数测试给定变量或者值的类型：

```
print(type("Hello world"))      --> string
print(type(10.4*3))             --> number
print(type(print))              --> function
print(type(type))               --> function
print(type(true))               --> boolean
print(type(nil))                --> nil
print(type(type(X)))            --> string
```

nil（空）

nil 类型表示一种没有任何有效值，它只有一个值 -- nil，例如打印一个没有赋值的变量，便会输出一个 nil 值：

```
> print(type(a))
nil
>
```

对于全局变量和 table，nil 还有一个"删除"作用，给全局变量或者 table 表里的变量赋一个 nil 值，等同于把它们删掉，执行下面代码就知：

```
tab1 = { key1 = "val1", key2 = "val2", "val3" }
for k, v in pairs(tab1) do
    print(k .. " - " .. v)
end

tab1.key1 = nil
for k, v in pairs(tab1) do
    print(k .. " - " .. v)
end
```

boolean（布尔）

boolean 类型只有两个可选值：true（真） 和 false（假），Lua 把 false 和 nil 看作是"假", 其他的都为"真":

```
print(type(true))
print(type(false))
print(type(nil))

if type(false) or type(nil) then
    print("false and nil are false!")
else
    print("other is true!")
end
```

以上代码执行结果如下：

```
$ lua test.lua
boolean
boolean
nil
false and nil are false!
```

number（数字）

Lua 默认只有一种 number 类型 -- double（双精度）类型（默认类型可以修改 luaconf.h 里的定义），以下几种写法都被看作是 number 类型：

```
print(type(2))
print(type(2.2))
print(type(0.2))
print(type(2e+1))
print(type(0.2e-1))
print(type(7.8263692594256e-06))
```

以上代码执行结果：

```
number
number
number
number
number
number
```

string（字符串）

字符串由一对双引号或单引号来表示。

```
string1 = "this is string1"
string2 = 'this is string2'
```

也可以用 2 个方括号 "[[]]" 来表示"一块"字符串。

```
html = [[
<html>
<head></head>
<body>
    <a href="http://www.w3cschool.cc/">w3cschool菜鸟教程</a>
</body>
</html>
]]
print(html)
```

以下代码执行结果为：

```
<html>
<head></head>
<body>
    <a href="http://www.w3cschool.cc/">w3cschool菜鸟教程</a>
</body>
</html>
```

在对一个数字字符串上进行算术操作时，Lua 会尝试将这个数字字符串转成一个数字:

```
> print("2" + 6)
8.0
> print("2" + "6")
8.0
> print("2 + 6")
2 + 6
> print("-2e2" * "6")
-1200.0
> print("error" + 1)
stdin:1: attempt to perform arithmetic on a string value
stack traceback:
    stdin:1: in main chunk
    [C]: in ?
>
```

以上代码中"error" + 1执行报错了，字符串连接使用的是 ..， 如：

```
> print("a" .. 'b')
ab
> print(157 .. 428)
157428
>
```

使用 # 来计算字符串的长度，放在字符串前面，如下实例：

```
> len = "www.w3cschool.cc"
> print(#len)
16
> print(#"www.w3cschool.cc")
16
>
```

table（表）

在 Lua 里，table 的创建是通过"构造表达式"来完成，最简单构造表达式是{}，用来创建一个空表。也可以在表里添加一些数据，直接初始化表:

```
-- 创建一个空的 table
local tbl1 = {}

-- 直接初始表
local tbl2 = {"apple", "pear", "orange", "grape"}
```

Lua 中的表（table）其实是一个"关联数组"（associative arrays），数组的索引可以是数字或者是字符串。

```
-- table_test.lua 脚本文件
a = {}
a["key"] = "value"
key = 10
a[key] = 22
a[key] = a[key] + 11
for k, v in pairs(a) do
    print(k .. " : " .. v)
end
```

脚本执行结果为：

```
$ lua table_test.lua
key : value
10 : 33
```

不同于其他语言的数组把 0 作为数组的初始索引，在 Lua 里表的默认初始索引一般以 1 开始。

```
-- table_test2.lua 脚本文件
local tbl = {"apple", "pear", "orange", "grape"}
for key, val in pairs(tbl) do
    print("Key", key)
end
```

脚本执行结果为：

```
$ lua table_test2.lua
Key    1
Key    2
Key    3
Key    4
```

table 不会固定长度大小，有新数据添加时 table 长度会自动增长，没初始的 table 都是 nil。

```
-- table_test3.lua 脚本文件
a3 = {}
for i = 1, 10 do
    a3[i] = i
end
a3["key"] = "val"
print(a3["key"])
print(a3["none"])
```

脚本执行结果为：

```
$ lua table_test3.lua
val
nil
```

function（函数）

在 Lua 中，函数是被看作是"第一类值（First-Class Value）"，函数可以存在变量里:

```
-- function_test.lua 脚本文件
function factorial1(n)
    if n == 0 then
        return 1
    else
        return n * factorial1(n - 1)
    end
end
print(factorial1(5))
factorial2 = factorial1
print(factorial2(5))
```

脚本执行结果为：

```
$ lua function_test.lua
120
120
```

function 可以以匿名函数（anonymous function）的方式通过参数传递:

```
-- function_test2.lua 脚本文件
function anonymous(tab, fun)
    for k, v in pairs(tab) do
        print(fun(k, v))
    end
end
tab = { key1 = "val1", key2 = "val2" }
anonymous(tab, function(key, val)
    return key .. " = " .. val
end)
```

脚本执行结果为：

```
$ lua function_test2.lua
key1 = val1
key2 = val2
```

thread（线程）

在 Lua 里，最主要的线程是协同程序（coroutine）。它跟线程（thread）差不多，拥有自己独立的栈、局部变量和指令指针，可以跟其他协同程序共享全局变量和其他大部分东西。

线程跟协程的区别：线程可以同时多个运行，而协程任意时刻只能运行一个，并且处于运行状态的协程只有被挂起（suspend）时才会暂停。

userdata（自定义类型）

userdata 是一种用户自定义数据，用于表示一种由应用程序或 C/C++ 语言库所创建的类型，可以将任意 C/C++ 的任意数据类型的数据（通常是 struct 和 指针）存储到 Lua 变量中调用。

Lua 变量

变量在使用前，必须在代码中进行声明，即创建该变量。编译程序执行代码之前编译器需要知道如何给语句变量开辟存储区，用于存储变量的值。

Lua 变量有三种类型：全局变量、局部变量、表中的域。

函数外的变量默认为全局变量，除非用 local 显示声明。函数内变量与函数的参数默认为局部变量。

局部变量的作用域为从声明位置开始到所在语句块结束（或者是直到下一个同名局部变量的声明）。

变量的默认值均为 nil。

```
-- test.lua 文件脚本 a =  5  --  全局变量  local b =  5  --  局部变量  function joke() c =  5
```

执行以上实例输出结果为：

```
$ lua test.lua nil  nil  6  6  5  6
```

赋值语句

赋值是改变一个变量的值和改变表域的最基本的方法。

```
a =  "hello"  ..  "world" t.n = t.n +  1
```

Lua可以对多个变量同时赋值，变量列表和值列表的各个元素用逗号分开，赋值语句右边的值会依次赋给左边的变量。

```
a, b =  10,  2*x <--> a=10; b=2*x
```

遇到赋值语句Lua会先计算右边所有的值然后再执行赋值操作，所以我们可以这样进行交换变量的值：

```
x, y = y, x -- swap 'x'  for  'y' a[i], a[j]  = a[j], a[i]  -- swap 'a[i]'  for  'a[i]'
```

当变量个数和值的个数不一致时，Lua会一直以变量个数为基础采取以下策略：

```
a. 变量个数 > 值的个数 按变量个数补足nil b. 变量个数 < 值的个数 多余的值会被忽略
```

例如：

```
a, b, c = 0, 1 print(a,b,c) --> 0 1 nil a, b = a+1, b+1, b+2 -- value of b+2 is
```

上面最后一个例子是一个常见的错误情况，注意：如果要对多个变量赋值必须依次对每个变量赋值。

```
a, b, c = 0, 0, 0 print(a,b,c) --> 0 0 0
```

多值赋值经常用来交换变量，或将函数调用返回给变量：

```
a, b = f()
```

f()返回两个值，第一个赋给a，第二个赋给b。

应该尽可能的使用局部变量，有两个好处：

- 1. 避免命名冲突。
- 2. 访问局部变量的速度比全局变量更快。

索引

对 table 的索引使用方括号 []。Lua 也提供了 . 操作。

```
t[i] t.i -- 当索引为字符串类型时的一种简化写法 gettable_event(t,i) -- 采用索引访问本质上是一个类
```

例如：

```
> site = {} > site["key"] = "www.w3cschool.cc" > print(site["key"]) www.w3cschool.c
```

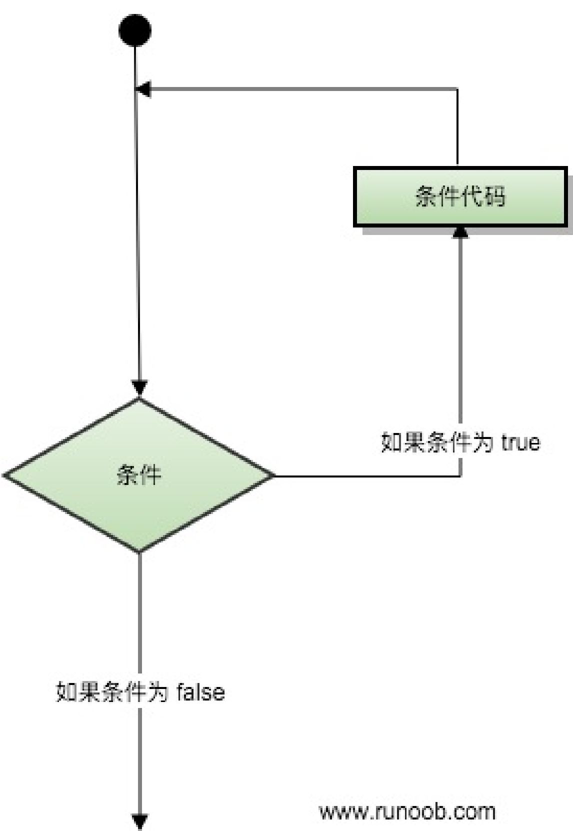
Lua 循环

很多情况下我们需要做一些有规律性的重复操作，因此在程序中就需要重复执行某些语句。

一组被重复执行的语句称之为循环体，能否继续重复，决定循环的终止条件。

循环结构是在一定条件下反复执行某段程序的流程结构，被反复执行的程序被称为循环体。

循环语句是由循环体及循环的终止条件两部分组成的。



Lua 语言提供了以下几种循环处理方式：

循环类型	描述
while 循环	在条件为 true 时，让程序重复地执行某些语句。执行语句前会先检查条件是否为 true。
for 循环	重复执行指定语句，重复次数可在 for 语句中控制。
Lua repeat...until	重复执行循环，直到 指定的条件为真时为止
循环嵌套	可以在循环内嵌套一个或多个循环语句（while、for、do..while）

循环控制语句

循环控制语句用于控制程序的流程， 以实现程序的各种结构方式。

Lua 支持以下循环控制语句：

控制语句	描述
break 语句	退出当前循环或语句，并开始脚本执行紧接着的语句。

无限循环

在循环体中如果条件永远为 true 循环语句就会永远执行下去，以下以 while 循环为例：

```
while( true ) do print("循环将永远执行下去") end
```

Lua while 循环

Lua 编程语言中 while 循环语句在判断条件为 true 时会重复执行循环体语句。

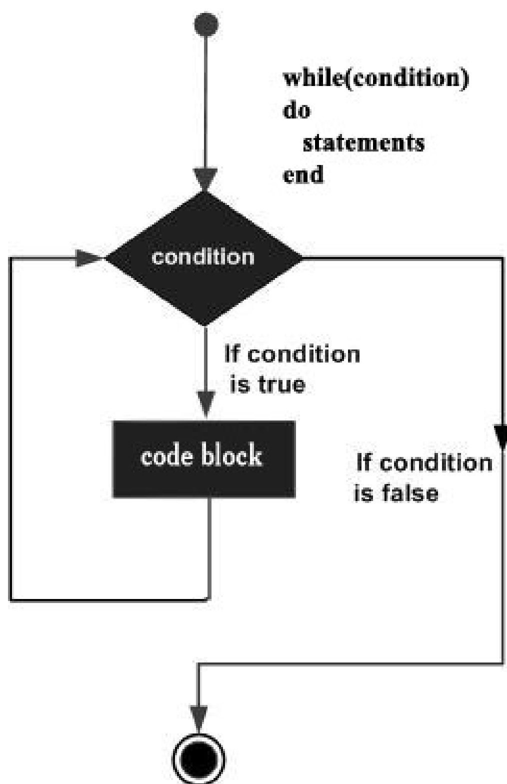
语法

Lua 编程语言中 while 循环语法：

```
while(condition)  do statements end
```

statements(循环体语句) 可以是一条或多条语句，**condition(条件)** 可以是任意表达式，在 **condition(条件)** 为 true 时执行循环体语句。

流程图如下：



在以上流程图中我们可以看出在**condition(条件)**为 false 时会跳过当前循环并开始脚本执行紧接着的语句。

实例

以下实例循环输出 a 的值：

```
a=10  while( a <  20  )  do  print("a 的值为:", a) a = a+1  end
```

执行以上代码，输出结果如下：

```
a 的值为:  10 a 的值为:  11 a 的值为:  12 a 的值为:  13 a 的值为:  14 a 的值为:  15 a 的值为:  1
```

Lua for 循环

Lua 编程语言中 for 循环语句可以重复执行指定语句，重复次数可在 for 语句中控制。

Lua 编程语言中 for语句有两大类：：

- 数值for循环
- 泛型for循环

数值for循环

Lua 编程语言中数值for循环语法格式:

```
for  var=exp1,exp2,exp3 do  <执行体>  end
```

var从exp1变化到exp2，每次变化以exp3为步长递增var，并执行一次"执行体"。exp3是可选的，如果不指定，默认为1。

实例

```
for i=1,f(x)  do  print(i)  end  for i=10,1,-1  do  print(i)  end
```

for的三个表达式在循环开始前一次性求值，以后不再进行求值。比如上面的f(x)只会在循环开始前执行一次，其结果用在后面的循环中。

验证如下:

```
#!/usr/local/bin/lua  function f(x)  print("function")  return x*2  end  for i=1,f(5)  do
```

以上实例输出结果为：

```
function  1  2  3  4  5  6  7  8  9  10
```

可以看到 函数f(x)只在循环开始前执行一次。

泛型for循环

泛型for循环通过一个迭代器函数来遍历所有值，类似java中的foreach语句。

Lua 编程语言中泛型for循环语法格式:

```
--打印数组a的所有值  for i,v in ipairs(a)  do  print(v)  end
```

i是数组索引值，v是对应索引的数组元素值。ipairs是Lua提供的一个迭代器函数，用来迭代数组。

实例

循环数组 days：

```
#!/usr/local/bin/lua days =  {"Suanday","Monday","Tuesday","Wednesday","Thursday","Friday
```

以上实例输出结果为：

```
Suanday  Monday  Tuesday  Wednesday  Thursday  Friday  Saturday
```

Lua repeat...until 循环

Lua 编程语言中 repeat...until 循环语句不同于 for 和 while循环，for 和 while循环d的条件语句在当前循环执行开始时判断，而 repeat...until 循环的条件语句在当前循环结束后判断。

语法

Lua 编程语言中 repeat...until 循环语法格式:

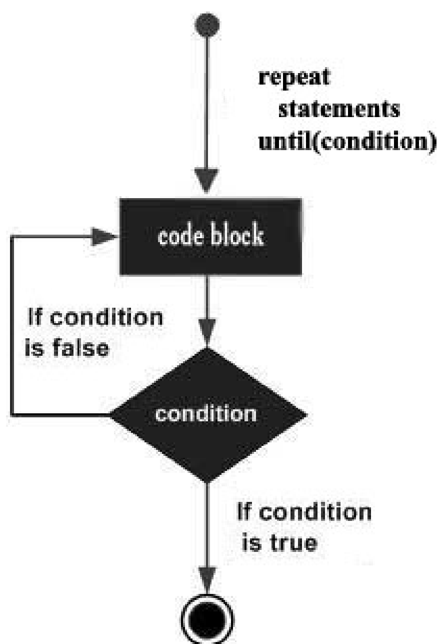
```
repeat
   statements while( condition )
```

repeat...until 是条件后行,所以repeat...until 的循环体里面至少要运行一次。

statements(循环体语句) 可以是一条或多条语句，**condition(条件)** 可以是任意表达式，在 **condition(条件)** 为 true 时执行循环体语句。

在**condition(条件)**为 false 时会跳过当前循环并开始脚本执行紧接着的语句。

Lua repeat...until 循环流程图如下：



实例

```
--[  变量定义  --] a =  10  --[  执行循环  --] repeat print("a的值为:", a) a = a +  1  until(
```

执行以上代码，程序输出结果为：

```
a的值为:  10 a的值为:  11 a的值为:  12 a的值为:  13 a的值为:  14 a的值为:  15
```

Lua 循环嵌套

Lua 编程语言中允许循环中嵌入循环。以下实例演示了 Lua 循环嵌套的应用。

语法

Lua 编程语言中 **for** 循环嵌套语法格式:

```
for init,max/min value, increment do  for init,max/min value, increment do statements end
```

Lua 编程语言中 **while** 循环嵌套语法格式:

```
while(condition)  do  while(condition)  do statements end statements end
```

Lua 编程语言中 **repeat...until** 循环嵌套语法格式:

```
repeat
   statements
   repeat
      statements until( condition )  until( condition )
```

除了以上同类型循环嵌套外，我们还可以使用不同的循环类型来嵌套，如 for 循环体中嵌套 while 循环。

实例

以下实例使用了for循环嵌套:

```
j =2  for i=2,10  do  for j=2,(i/j)  ,  2  do  if(not(i%j))  then  break  end  if(j >  (i
```

以上代码执行结果为：

```
i 的值为：  8 i 的值为：  9 i 的值为：  10
```

Lua break 语句

Lua 编程语言 break 语句插入在循环体中，用于退出当前循环或语句，并开始脚本执行紧接着的语句。

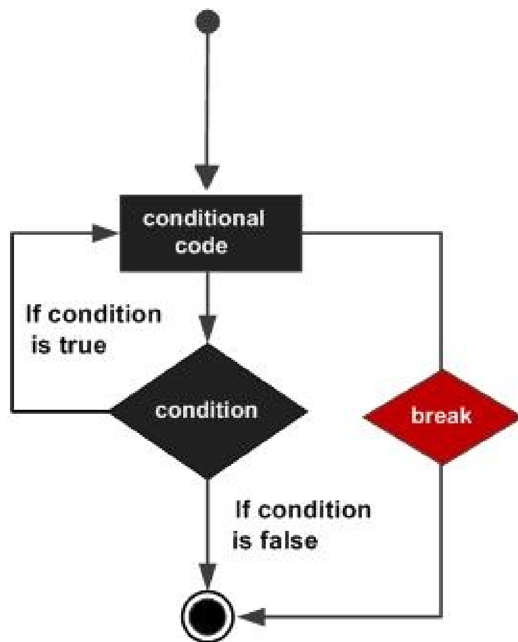
如果你使用循环嵌套，break语句将停止最内层循环的执行，并开始执行的外层的循环语句。

语法

Lua 编程语言中 **break** 语句语法格式:

```
break
```

流程图：



实例

以下实例执行 while 循环，在变量 a 小于 20 时输出 a 的值，并在 a 大于 15 时终止执行循环：

```
--[ 定义变量 --] a = 10 --[ while 循环 --] while( a < 20 ) do print("a 的值为:",
```

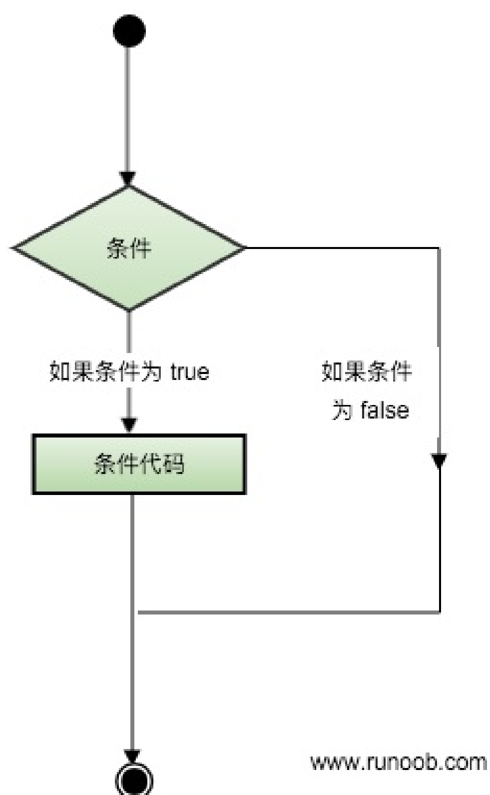
以上代码执行结果如下：

```
a 的值为: 10 a 的值为: 11 a 的值为: 12 a 的值为: 13 a 的值为: 14 a 的值为: 15
```

Lua 流程控制

Lua 编程语言流程控制语句通过程序设定一个或多个条件语句来设定。在条件为 true 时执行指定程序代码，在条件为 false 时执行其他指定代码。

以下是典型的流程控制流程图：



控制结构的条件表达式结果可以是任何值，Lua认为false和nil为假，true 和非nil为真。

要注意的是Lua中 0 为 true：

```
--[ 0 为true ] if(0) then print("0 为真") end
```

以上代码输出结果为：

```
0 为真
```

Lua 提供了以下控制结构语句：

语句	描述
if 语句	**if** 语句 由一个布尔表达式作为条件判断，其后紧跟其他语句组成。
if...else 语句	**if** 语句 可以与 **else** 语句搭配使用, 在 if 条件表达式为 false 时执行 else 语句代码。
if 嵌套语句	你可以在**if** 或 **else if**中使用一个或多个 **if** 或 **else if** 语句 。

Lua if 语句

Lua **if** 语句 由一个布尔表达式作为条件判断，其后紧跟其他语句组成。

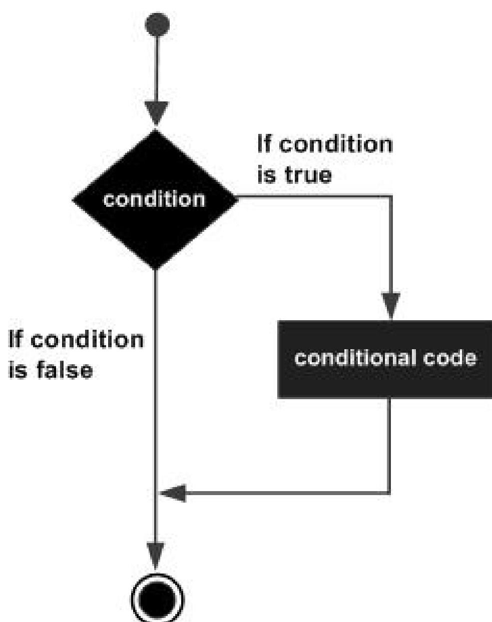
Lua if 语句语法格式如下：

```
if(布尔表达式) then --[ 在布尔表达式为 true 时执行的语句 --] end
```

在布尔表达式为 true 时会if中的代码块会被执行，在布尔表达式为 false 时，紧跟在 if 语句 end 之后的代码会被执行。

Lua认为false和nil为假，true 和非nil为真。要注意的是Lua中 0 为 true。

if 语句流程图如下：



实例

以下实例用于判断变量 a 的值是否小于 20：

```
--[ 定义变量 --] a = 10; --[ 使用 if 语句 --] if( a < 20 ) then --[ if 条件为
```

以上代码执行结果如下：

```
a 小于 20 a 的值为: 10
```

Lua if...else 语句

if...else 语句

Lua if 语句可以与 else 语句搭配使用, 在 if 条件表达式为 false 时执行 else 语句代码块。

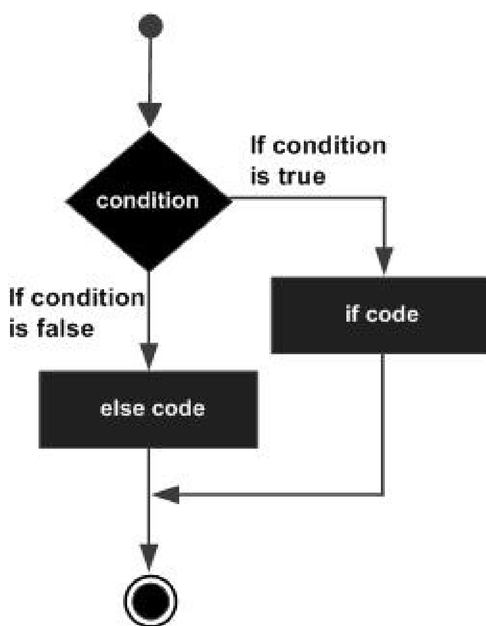
Lua if...else 语句语法格式如下：

```
if(布尔表达式) then --[ 布尔表达式为 true 时执行该语句块 --] else --[ 布尔表达式为 fals
```

在布尔表达式为 true 时会if中的代码块会被执行，在布尔表达式为 false 时，else 的代码块会被执行。

Lua认为false和nil为假，true 和非nil为真。要注意的是Lua中 0 为 true。

if 语句流程图如下：



实例

以下实例用于判断变量 a 的值：

```
--[ 定义变量 --] a = 100; --[ 检查条件 --] if( a < 20 ) then --[ if 条件为 true
```

以上代码执行结果如下：

```
a 大于  20 a 的值为  :  100
```

if...else if...else 语句

Lua if 语句可以与 else if...else 语句搭配使用, 在 if 条件表达式为 false 时执行 else if...else 语句代码块，用于检测多个条件语句。

Lua if...else if...else 语句语法格式如下：

```
if(  布尔表达式  1)  then  --[  在布尔表达式  1  为  true  时执行该语句块  --]  else  if(  布尔表
```

实例

以下实例对变量 a 的值进行判断：

```
--[  定义变量  --] a =  100  --[  检查布尔条件  --]  if( a ==  10  )  then  --[  如果条件为  t
```

以上代码执行结果如下：

```
没有匹配  a  的值  a  的真实值为:  100
```

Lua if 嵌套语句

if...else 语句

Lua if 语句允许嵌套, 这就意味着你可以在一个 if 或 else if 语句中插入其他的 if 或 else if 语句。

Lua if 嵌套语句语法格式如下：

```
if( 布尔表达式 1) then --[ 布尔表达式 1 为 true 时执行该语句块 --] if(布尔表达式 2)
```

你可以用同样的方式嵌套 **else if...else** 语句。

实例

以下实例用于判断变量 a 和 b 的值：

```
--[ 定义变量 --] a = 100; b = 200; --[ 检查条件 --] if( a == 100 ) then --[ if
```

以上代码执行结果如下：

```
a 的值为 100 b 的值为 200 a 的值为 : 100 b 的值为 : 200
```

Lua 函数

在Lua中，函数是对语句和表达式进行抽象的主要方法。既可以用来处理一些特殊的工作，也可以用来计算一些值。

Lua 提供了许多的内建函数，你可以很方便的在程序中调用它们，如print()函数可以将传入的参数打印在控制台上。

Lua 函数主要有两种用途：

- 1.完成指定的任务，这种情况下函数作为调用语句使用；
- 2.计算并返回值，这种情况下函数作为赋值语句的表达式使用。

函数定义

Lua 编程语言函数定义格式如下：

```
optional_function_scope function function_name( argument1, argument2, argument3..., argum
    function_body
    return result_params_comma_separated
end
```

解析：

- **optional_function_scope**

local

- **function_name:**

- **argument1, argument2, argument3..., argumentn:**

- **function_body:**

- **result_params_comma_separated:**

实例

以下实例定义了函数 **max()**，参数为 num1, num2，用于比较两值的大小，并返回最大值：

```
--[[ 函数返回两个值的最大值 --]]
function max(num1, num2)

   if (num1 > num2) then
      result = num1;
   else
      result = num2;
   end

   return result;
end
-- 调用函数
print("两值比较最大值为 ",max(10,4))
print("两值比较最大值为 ",max(5,6))
```

以上代码执行结果为：

```
两值比较最大值为     10
两值比较最大值为     6
```

Lua 中我们可以将函数作为参数传递给函数，如下实例：

```
myprint = function(param)
   print("这是打印函数 -   ##",param,"##")
end

function add(num1,num2,functionPrint)
   result = num1 + num2
   -- 调用传递的函数参数
   functionPrint(result)
end
myprint(10)
-- myprint 函数作为参数传递
add(2,5,myprint)
```

以上代码执行结果为：

```
这是打印函数 -   ##    10    ##
这是打印函数 -   ##    7    ##
```

多返回值

Lua函数可以返回多个结果值，比如string.find，其返回匹配串"开始和结束的下标"（如果不存在匹配串返回nil）。

```
> s, e = string.find("w3cschool菜鸟教程：www.w3cschool.cc", "菜鸟教程")
> print(s,e)
10    21
```

Lua函数中，在return后列出要返回的值得列表即可返回多值，如：

```
function maximum (a)
    local mi = 1             -- 最大值索引
    local m = a[mi]          -- 最大值
    for i,val in ipairs(a) do
       if val > m then
           mi = i
           m = val
       end
    end
    return m, mi
end

print(maximum({8,10,23,12,5}))
```

以上代码执行结果为：

```
23    3
```

可变参数

Lua函数可以接受可变数目的参数，和C语言类似在函数参数列表中使用三点（...) 表示函数有可变的参数。

Lua将函数的参数放在一个叫arg的表中，**#arg** 表示传入参数的个数。

例如，我们计算几个数的平均值：

```
function average(...)
   result = 0
   local arg={...}
   for i,v in ipairs(arg) do
      result = result + v
   end
   print("总共传入 " .. #arg .. " 个数")
   return result/#arg
end

print("平均值为",average(10,5,3,4,5,6))
```

以上代码执行结果为：

```
总共传入 6 个数
平均值为    5.5
```

Lua 运算符

运算符是一个特殊的符号，用于告诉解释器执行特定的数学或逻辑运算。Lua提供了以下几种运算符类型：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 其他运算符

算术运算符

下表列出了 Lua 语言中的常用算术运算符，设定 A 的值为10，B 的值为 20：

操作符	描述	实例
+	加法	A + B 输出结果 30
-	减法	A - B 输出结果 -10
*	乘法	A * B 输出结果 200
/	除法	B / A w输出结果 2
%	取余	B % A 输出结果 0
^	乘幂	A^2 输出结果 100
-	负号	-A 输出结果v -10

实例

我们可以通过以下实例来更加透彻的理解算术运算符的应用：

```
a = 21
b = 10
c = a + b
print("Line 1 - c 的值为 ", c )
c = a - b
print("Line 2 - c 的值为 ", c )
c = a * b
print("Line 3 - c 的值为 ", c )
c = a / b
print("Line 4 - c 的值为 ", c )
c = a % b
print("Line 5 - c 的值为 ", c )
c = a^2
print("Line 6 - c 的值为 ", c )
c = -a
print("Line 7 - c 的值为 ", c )
```

以上程序执行结果为：

```
Line 1 - c 的值为     31
Line 2 - c 的值为     11
Line 3 - c 的值为     210
Line 4 - c 的值为     2.1
Line 5 - c 的值为     1
Line 6 - c 的值为     441
Line 7 - c 的值为     -21
```

关系运算符

下表列出了 Lua 语言中的常用关系运算符，设定 A 的值为10，B 的值为 20：

操作符	描述	实例
==	等于，检测两个值是否相等，相等返回 true，否则返回 false	(A == B) 为 false。
~=	不等于，检测两个值是否相等，相等返回 false，否则返回 true<	(A ~= B) 为 true。
>	大于，如果左边的值大于右边的值，返回 true，否则返回 false	(A > B) 为 false。
<	小于，如果左边的值大于右边的值，返回 false，否则返回 true	(A < B) 为 true。
>=	大于等于，如果左边的值大于等于右边的值，返回 true，否则返回 false	(A >= B) is not true.
<=	小于等于， 如果左边的值小于等于右边的值，返回 true，否则返回 false	(A <= B) is true.

实例

我们可以通过以下实例来更加透彻的理解关系运算符的应用：

```
a = 21
b = 10

if( a == b )
then
   print("Line 1 - a 等于 b" )
else
   print("Line 1 - a 不等于 b" )
end

if( a ~= b )
then
   print("Line 2 - a 不等于 b" )
else
   print("Line 2 - a 等于 b" )
end

if ( a < b )
then
   print("Line 3 - a 小于 b" )
else
   print("Line 3 - a 大于等于 b" )
end

if ( a > b )
then
   print("Line 4 - a 大于 b" )
else
   print("Line 5 - a 小于等于 b" )
end

-- 修改 a 和 b 的值
a = 5
b = 20
if ( a <= b )
then
   print("Line 5 - a 小于等于  b" )
end

if ( b >= a )
then
   print("Line 6 - b 大于等于 a" )
end
```

以上程序执行结果为：

```
Line 1 - a 不等于 b
Line 2 - a 不等于 b
Line 3 - a 大于等于 b
Line 4 - a 大于 b
Line 5 - a 小于等于  b
Line 6 - b 大于等于 a
```

逻辑运算符

下表列出了 Lua 语言中的常用逻辑运算符，设定 A 的值为10，B 的值为 20：

操作符	描述	实例
and	逻辑与操作符。 如果两边的操作都为 true 则条件为 true。	(A and B) 为 false。
or	逻辑或操作符。 如果两边的操作任一一个为 true 则条件为 true。	(A or B) 为 true。
not	逻辑非操作符。与逻辑运算结果相反，如果条件为 true，逻辑非为 false。	!(A and B) 为 false。

实例

我们可以通过以下实例来更加透彻的理解逻辑运算符的应用：

```
a = 5
b = 20

if ( a and b )
then
   print("Line 1 - 条件为 true" )
end

if ( a or b )
then
   print("Line 2 - 条件为 true" )
end

-- 修改 a 和 b 的值
a = 0
b = 10

if ( a and b )
then
   print("Line 3 - 条件为 true" )
else
   print("Line 3 - 条件为 false" )
end

if ( not( a and b) )
then
   print("Line 4 - 条件为 true" )
else
   print("Line 3 - 条件为 false" )
end
```

以上程序执行结果为：

```
Line 1 - 条件为 true
Line 2 - 条件为 true
Line 3 - 条件为 true
Line 3 - 条件为 false
```

其他运算符

下表列出了 Lua 语言中的连接运算符与计算表或字符串长度的运算符：

操作符	描述	实例
..	连接两个字符串	a..b ，其中 a 为 "Hello " ， b 为 "World", 输出结果为 "Hello World"。
#	一元运算符，返回字符串或表的长度。	#"Hello" 返回 5

实例

我们可以通过以下实例来更加透彻的理解连接运算符与计算表或字符串长度的运算符的应用：

```
a = "Hello "
b = "World"

print("连接字符串 a 和 b ", a..b )

print("b 字符串长度 ",#b )

print("字符串 Test 长度 ",#"Test" )

print("w3cschool菜鸟教程网址长度 ",#"www.w3cschool.cc" )
```

以上程序执行结果为：

```
连接字符串 a 和 b     Hello World
b 字符串长度     5
字符串 Test 长度     4
w3cschool菜鸟教程网址长度     16
```

运算符优先级

从高到低的顺序：

```
^
not    - (unary)
*      /
+      -
..
<      >      <=      >=      ~=      ==
and
or
```

除了^和..外所有的二元运算符都是左连接的。

```
a+i < b/2+1          <-->       (a+i) < ((b/2)+1)
5+x^2*8              <-->       5+((x^2)*8)
a < y and y <= z     <-->       (a < y) and (y <= z)
-x^2                 <-->       -(x^2)
x^y^z                <-->       x^(y^z)
```

实例

我们可以通过以下实例来更加透彻的了解 Lua 语言运算符的优先级：

```
a = 20
b = 10
c = 15
d = 5

e = (a + b) * c / d;-- ( 30 * 15 ) / 5
print("(a + b) * c / d 运算值为  :",e )

e = ((a + b) * c) / d; -- (30 * 15 ) / 5
print("((a + b) * c) / d 运算值为 :",e )

e = (a + b) * (c / d);-- (30) * (15/5)
print("(a + b) * (c / d) 运算值为 :",e )

e = a + (b * c) / d;  -- 20 + (150/5)
print("a + (b * c) / d 运算值为   :",e )
```

以上程序执行结果为：

```
(a + b) * c / d 运算值为  :    90.0
((a + b) * c) / d 运算值为 :    90.0
(a + b) * (c / d) 运算值为 :    90.0
a + (b * c) / d 运算值为   :    50.0
```

Lua 字符串

字符串或串(String)是由数字、字母、下划线组成的一串字符。

Lua 语言中字符串可以使用以下三种方式来表示：

- 单引号间的一串字符。
- 双引号间的一串字符。
- [[和]]间的一串字符。

以上三种方式的字符串实例如下：

```
string1 =  "Lua"  print("\"字符串 1 是\"",string1) string2 =  'w3cschool.cc'  print("字符串
```

以上代码执行输出结果为：

```
"字符串 1 是"  Lua  字符串  2  是 w3cschool.cc 字符串  3  是  "Lua 教程"
```

转义字符用于表示不能直接显示的字符，比如后退键，回车键，等。如在字符串转换双引号可以使用 "\""。

所有的转义字符和所对应的意义：

转义字符	意义	ASCII码值（十进制）
\a	响铃(BEL)	007
\b	退格(BS)，将当前位置移到前一列	008
\f	换页(FF)，将当前位置移到下页开头	012
\n	换行(LF)，将当前位置移到下一行开头	010
\r	回车(CR)，将当前位置移到本行开头	013
\t	水平制表(HT) （跳到下一个TAB位置）	009
\v	垂直制表(VT)	011
\\	代表一个反斜线字符''\'	092
\'	代表一个单引号（撇号）字符	039
\"	代表一个双引号字符	034
\0	空字符(NULL)	000
\ddd	1到3位八进制数所代表的任意字符	三位八进制
\xhh	1到2位十六进制所代表的任意字符	二位十六进制

字符串操作

Lua 提供了很多的方法来支持字符串的操作：

方法	用途	
string.upper(argument):	字符串全部转为大写字母。	
string.lower(argument):	字符串全部转为小写字母。	
string.gsub(mainString,findString,replaceString,num)	在字符串中替换,mainString 为要替换的字符串，findString 为被替换的字符，replaceString 要替换的字符，num 替换次数（可以忽略，则全部替换），	`string.gsub`
	在一个指定的目标字符串中	

string.strfind (str, substr, [init, [end]])	搜索指定的内容(第三个参数为索引),返回其具体位置。不存在则返回 nil。	`string.find`
string.reverse(arg)	字符串反转	`string.reve`
string.format(...)	返回一个类似printf的格式化字符串	`string.form`
string.char(arg) 和 string.byte(arg[,int])	char 将整型数字转成字符并连接, byte 转换字符为整数值(可以指定某个字符，默认第一个字符)。	`string.char` `string.byte` `string.byte`
string.len(arg)	计算字符串长度。	`string.len(`
string.rep(string, n))	返回字符串string的n个拷贝	`string.rep(`
..	链接两个字符串	`print("www.`

字符串大小写转换

以下实例演示了如何对字符串大小写进行转换：

```
string1 =  "Lua";  print(string.upper(string1))  print(string.lower(string1))
```

以上代码执行结果为：

```
LUA
lua
```

字符串查找与反转

以下实例演示了如何对字符串进行查找与反转操作：

```
string  =  "Lua Tutorial"  --  查找字符串  print(string.find(string,"Tutorial")) reversedSt
```

以上代码执行结果为：

```
5  12  新字符串为 lairotuT auL
```

字符串格式化

以下实例演示了如何对字符串进行格式化操作：

```
string1 =  "Lua" string2 =  "Tutorial" number1 =  10 number2 =  20  --  基本字符串格式化  pr
```

以上代码执行结果为：

```
基本格式化  Lua  Tutorial  日期格式化  02/01/2014  0.3333
```

字符与整数相互转换

以下实例演示了字符与整数相互转换：

```
--  字符转换  --  转换第一个字符  print(string.byte("Lua"))  --  转换第三个字符  print(string.b
```

以上代码执行结果为：

```
76  97  97  117  117 a
```

其他常用函数

以下实例演示了其他字符串操作，如计算字符串长度，字符串连接，字符串复制等：

```
string1 =  "www." string2 =  "w3cschool" string3 =  ".cc"  --  使用  ..  进行字符串连接  pri
```

以上代码执行结果为：

```
连接字符串 www.w3cschool.cc 字符串长度  9 w3cschoolw3cschool
```

Lua 数组

数组，就是相同数据类型的元素按一定顺序排列的集合，可以是一维数组和多维数组。

Lua 数组的索引键值可以使用整数表示，数组的大小不是固定的。

一维数组

一维数组是最简单的数组，其逻辑结构是线性表。一维数组可以用for循环出数组中的元素，如下实例：

```
array = {"Lua", "Tutorial"}

for i= 0, 2 do
   print(array[i])
end
```

以上代码执行输出结果为：

```
nil
Lua
Tutorial
```

正如你所看到的，我们可以使用整数索引来访问数组元素，如果知道的索引没有值则返回nil。

在 Lua 索引值是以 1 为起始，但你也可以指定 0 开始。

除此外我们还可以以负数为数组索引值：

```
array = {}

for i= -2, 2 do
   array[i] = i *2
end

for i = -2,2 do
   print(array[i])
end
```

以上代码执行输出结果为：

```
-4
-2
0
2
4
```

多维数组

多维数组即数组中包含数组或一维数组的索引键对应一个数组。

以下是一个三行三列的阵列多维数组：

```
-- 初始化数组
array = {}
for i=1,3 do
   array[i] = {}
      for j=1,3 do
         array[i][j] = i*j
      end
end

-- 访问数组
for i=1,3 do
   for j=1,3 do
      print(array[i][j])
   end
end
```

以上代码执行输出结果为：

```
1
2
3
2
4
6
3
6
9
```

不同索引键的三行三列阵列多维数组：

```
-- 初始化数组
array = {}
maxRows = 3
maxColumns = 3
for row=1,maxRows do
   for col=1,maxColumns do
      array[row*maxColumns +col] = row*col
   end
end

-- 访问数组
for row=1,maxRows do
   for col=1,maxColumns do
      print(array[row*maxColumns +col])
   end
end
```

以上代码执行输出结果为：

```
1
2
3
2
4
6
3
6
9
```

正如你所看到的，以上的实例中，数组设定了指定的索引值，这样可以避免出现 nil 值，有利于节省内存空间。

Lua 迭代器

迭代器（iterator）是一种对象，它能够用来遍历标准模板库容器中的部分或全部元素，每个迭代器对象代表容器中的确定的地址

在Lua中迭代器是一种支持指针类型的结构，它可以遍历集合的每一个元素。

泛型 for 迭代器

泛型 for 在自己内部保存迭代函数，实际上它保存三个值：迭代函数、状态常量、控制变量。

泛型 for 迭代器提供了集合的 key/value 对，语法格式如下：

```
for k, v in pairs(t) do
    print(k, v)
end
```

上面代码中，k, v为变量列表；pair(t)为表达式列表。

查看以下实例:

```
array = {"Lua", "Tutorial"}

for key,value in ipairs(array)
do
   print(key, value)
end
```

以上代码执行输出结果为：

```
1  Lua
2  Tutorial
```

以上实例中我们使用了 Lua 默认提供的迭代函数 ipairs。

下面我们看看范性for的执行过程：

- 首先，初始化，计算in后面表达式的值，表达式应该返回范性for需要的三个值：迭代函数、状态常量、控制变量；与多值赋值一样，如果表达式返回的结果个数不足三个会自动用nil补足，多出部分会被忽略。
- 第二，将状态常量和控制变量作为参数调用迭代函数（注意：对于for结构来说，状态常量没有用处，仅仅在初始化时获取他的值并传递给迭代函数）。
- 第三，将迭代函数返回的值赋给变量列表。
- 第四，如果返回的第一个值为nil循环结束，否则执行循环体。

- 第五，回到第二步再次调用迭代函数

。在Lua中我们常常使用函数来描述迭代器，每次调用该函数就返回集合的下一个元素。Lua的迭代器包含以下两种类型：

- 无状态的迭代器
- 多状态的迭代器

无状态的迭代器

无状态的迭代器是指不保留任何状态的迭代器，因此在循环中我们可以利用无状态迭代器避免创建闭包花费额外的代价。

每一次迭代，迭代函数都是用两个变量（状态常量和控制变量）的值作为参数被调用，一个无状态的迭代器只利用这两个值可以获取下一个元素。

这种无状态迭代器的典型的简单的例子是ipairs，他遍历数组的每一个元素。

以下实例我们使用了一个简单的函数来实现迭代器，实现 数字 n 的平方：

```
function square(iteratorMaxCount,currentNumber)
   if currentNumber<iteratorMaxCount
   then
      currentNumber = currentNumber+1
   return currentNumber, currentNumber*currentNumber
   end
end

for i,n in square,3,0
do
   print(i,n)
end
```

以上实例输出结果为：

```
1    1
2    4
3    9
```

迭代的状态包括被遍历的表（循环过程中不会改变的状态常量）和当前的索引下标（控制变量），ipairs和迭代函数都很简单，我们在Lua中可以这样实现：

```
function iter (a, i)
    i = i + 1
    local v = a[i]
    if v then
       return i, v
    end
end

function ipairs (a)
    return iter, a, 0
end
```

当Lua调用ipairs(a)开始循环时，他获取三个值：迭代函数iter、状态常量a、控制变量初始值0；然后Lua调用iter(a,0)返回1,a[1]（除非a[1]=nil）；第二次迭代调用iter(a,1)返回2,a[2]……直到第一个非nil元素。

多状态的迭代器

很多情况下，迭代器需要保存多个状态信息而不是简单的状态常量和控制变量，最简单的方法是使用闭包，还有一种方法就是将所有的状态信息封装到table内，将table作为迭代器的状态常量，因为这种情况下可以将所有的信息存放在table内，所以迭代函数通常不需要第二个参数。

以下实例我们创建了自己的迭代器：

```
array = {"Lua", "Tutorial"}

function elementIterator (collection)
   local index = 0
   local count = #collection
   -- 闭包函数
   return function ()
      index = index + 1
      if index <= count
      then
         --  返回迭代器的当前元素
         return collection[index]
      end
   end
end

for element in elementIterator(array)
do
   print(element)
end
```

以上实例输出结果为：

```
Lua
Tutorial
```

以上实例中我们可以看到，elementIterator 内使用了闭包函数，实现计算集合大小并输出各个元素。

Lua table(表)

table 是 Lua 的一种数据结构用来帮助我们创建不同的数据类型，如：数字、字典等。

Lua table 使用关联型数组，你可以用任意类型的值来作数组的索引，但这个值不能是 nil。

Lua table 是不固定大小的，你可以根据自己需要进行扩容。

Lua也是通过table来解决模块（module）、包（package）和对象（Object）的。 例如string.format表示使用"format"来索引table string。

table(表)的构造

构造器是创建和初始化表的表达式。表是Lua特有的功能强大的东西。最简单的构造函数是{}，用来创建一个空表。可以直接初始化数组:

```
-- 初始化表
mytable = {}

-- 指定值
mytable[1]= "Lua"

-- 移除引用
mytable = nil
-- lua 垃圾回收会释放内存
```

当我们为 table a 并设置元素，然后将 a 赋值给 b，则 a 与 b 都指向同一个内存。如果 a 设置为 nil ，则 b 同样能访问 table 的元素。如果没有指定的变量指向a，Lua的垃圾回收机制会清理相对应的内存。

以下实例演示了以上的描述情况：

```
-- 简单的 table
mytable = {}
print("mytable 的类型是 ",type(mytable))

mytable[1]= "Lua"
mytable["wow"] = "修改前"
print("mytable 索引为 1 的元素是 ", mytable[1])
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

-- alternatetable和mytable的是指同一个 table
alternatetable = mytable

print("alternatetable 索引为 1 的元素是 ", alternatetable[1])
print("mytable 索引为 wow 的元素是 ", alternatetable["wow"])

alternatetable["wow"] = "修改后"

print("mytable 索引为 wow 的元素是 ", mytable["wow"])

-- 释放变量
alternatetable = nil
print("alternatetable 是 ", alternatetable)

-- mytable 仍然可以访问
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

mytable = nil
print("mytable 是 ", mytable)
```

以上代码执行结果为：

```
mytable 的类型是 	table
mytable 索引为 1 的元素是 	Lua
mytable 索引为 wow 的元素是 	修改前
alternatetable 索引为 1 的元素是 	Lua
mytable 索引为 wow 的元素是 	修改前
mytable 索引为 wow 的元素是 	修改后
alternatetable 是 	nil
mytable 索引为 wow 的元素是 	修改后
mytable 是 	nil
```

Table 操作

以下列出了 Table 操作常用的方法：

方法	用途
table.concat (table [, step [, start [, end]]]):	concat是concatenate(连锁, 连接)的缩写. table.concat()函数列出参数中指定table的数组部分从start位置到end位置的所有元素, 元素间以指定的分隔符(sep)隔开。
table.insert (table, [pos,] value):	在table的数组部分指定位置(pos)插入值为value的一个元素. pos参数可选, 默认为数组部分末尾.
table.maxn (table)	指定table中所有正数key值中最大的key值. 如果不存在key值为正数的元素, 则返回0。(**Lua5.2**之后该方法已经不存在了,本文使用了自定义函数实现)
table.remove (table [, pos])	返回table数组部分位于pos位置的元素. 其后的元素会被前移. pos参数可选, 默认为table长度, 即从最后一个元素删起。
table.sort (table [, comp])	对给定的table进行升序排序。

接下来我们来看下这几个方法的实例。

Table 连接

我们可以使用 concat() 方法来连接两个 table:

```
fruits = {"banana","orange","apple"}
-- 返回 table 连接后的字符串
print("连接后的字符串 ",table.concat(fruits))

-- 指定连接字符
print("连接后的字符串 ",table.concat(fruits,", "))

-- 指定索引来连接 table
print("连接后的字符串 ",table.concat(fruits,", ", 2,3))
```

执行以上代码输出结果为：

```
连接后的字符串     bananaorangeapple
连接后的字符串     banana, orange, apple
连接后的字符串     orange, apple
```

插入和移除

以下实例演示了 table 的插入和移除操作:

```
fruits = {"banana","orange","apple"}

-- 在末尾插入
table.insert(fruits,"mango")
print("索引为 4 的元素为 ",fruits[4])

-- 在索引为 2 的键处插入
table.insert(fruits,2,"grapes")
print("索引为 2 的元素为 ",fruits[2])

print("最后一个元素为 ",fruits[5])
table.remove(fruits)
print("移除后最后一个元素为 ",fruits[5])
```

执行以上代码输出结果为：

```
索引为 4 的元素为      mango
索引为 2 的元素为      grapes
最后一个元素为      mango
移除后最后一个元素为      nil
```

Table 排序

以下实例演示了 sort() 方法的使用，用于对 Table 进行排序：

```
fruits = {"banana","orange","apple","grapes"}
print("排序前")
for k,v in ipairs(fruits) do
    print(k,v)
end

table.sort(fruits)
print("排序后")
for k,v in ipairs(fruits) do
    print(k,v)
end
```

执行以上代码输出结果为：

```
排序前
1    banana
2    orange
3    apple
4    grapes
排序后
1    apple
2    banana
3    grapes
4    orange
```

Table 最大值

table.maxn 在 Lua5.2 之后该方法已经不存在了，我们定义了 table_maxn 方法来实现。

以下实例演示了如何获取 table 中的最大值：

```
function table_maxn(t)
    local mn = 0
    for k, v in pairs(t) do
        if mn < k then
            mn = k
        end
    end
    return mn
end
tbl = {[1] = "a", [2] = "b", [3] = "c", [26] = "z"}
print("tbl 长度 ", #tbl)
print("tbl 最大值 ", table_maxn(tbl))
```

执行以上代码输出结果为：

```
tbl 长度     3
tbl 最大值     26
```

Lua 模块与包

模块类似于一个封装库，从 Lua 5.1 开始，Lua 加入了标准的模块管理机制，可以把一些公用的代码放在一个文件里，以 API 接口的形式在其他地方调用，有利于代码的重用和降低代码耦合度。

Lua 的模块是由变量、函数等已知元素组成的 table，因此创建一个模块很简单，就是创建一个 table，然后把需要导出的常量、函数放入其中，最后返回这个 table 就行。以下为创建自定义模块 module.lua，文件代码格式如下：

```
-- 文件名为 module.lua
-- 定义一个名为 module 的模块
module = {}

-- 定义一个常量
module.constant = "这是一个常量"

-- 定义一个函数
function module.func1()
    io.write("这是一个公有函数！\n")
end

local function func2()
    print("这是一个私有函数！")
end

function module.func3()
    func2()
end

return module
```

由上可知，模块的结构就是一个 table 的结构，因此可以像操作调用 table 里的元素那样来操作调用模块里的常量或函数。

上面的 func2 声明为程序块的局部变量，即表示一个私有函数，因此是不能从外部访问模块里的这个私有函数，必须通过模块里的公有函数来调用.

require 函数

Lua提供了一个名为require的函数用来加载模块。要加载一个模块，只需要简单地调用就可以了。例如：

```
require("<模块名>")
```

或者

```
require "<模块名>"
```

执行 require 后会返回一个由模块常量或函数组成的 table，并且还会定义一个包含该 table 的全局变量。

```
-- test_module.php 文件
-- module 模块为上文提到到 module.lua
require("module")

print(module.constant)

module.func3()
```

以上代码执行结果为：

```
这是一个常量
这是一个私有函数！
```

或者给加载的模块定义一个别名变量，方便调用：

```
-- test_module2.php 文件
-- module 模块为上文提到到 module.lua
-- 别名变量 m
local m = require("module")

print(m.constant)

m.func3()
```

以上代码执行结果为：

```
这是一个常量
这是一个私有函数！
```

加载机制

对于自定义的模块，模块文件不是放在哪个文件目录都行，函数 require 有它自己的文件路径加载策略，它会尝试从 Lua 文件或 C 程序库中加载模块。

require 用于搜索 Lua 文件的路径是存放在全局变量 package.path 中，当 Lua 启动后，会以环境变量 LUA_PATH 的值来初始这个环境变量。如果没有找到该环境变量，则使用一个编译时定义的默认路径来初始化。

当然，如果没有 LUA_PATH 这个环境变量，也可以自定义设置，在当前用户根目录下打开 .profile 文件（没有则创建，打开 .bashrc 文件也可以），例如把 "~/lua/" 路径加入 LUA_PATH 环境变量里：

```
#LUA_PATH
export LUA_PATH="~/lua/?.lua;;"
```

文件路径以 ";" 号分隔，最后的 2 个 ";;" 表示新加的路径后面加上原来的默认路径。

接着，更新环境变量参数，使之立即生效。

```
source ~/.profile
```

这时假设 package.path 的值是：

```
/Users/dengjoe/lua/?.lua;./?.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/
```

那么调用 require("module") 时就会尝试打开以下文件目录去搜索目标。

```
/Users/dengjoe/lua/module.lua;
./module.lua
/usr/local/share/lua/5.1/module.lua
/usr/local/share/lua/5.1/module/init.lua
/usr/local/lib/lua/5.1/module.lua
/usr/local/lib/lua/5.1/module/init.lua
```

如果找过目标文件，则会调用 package.loadfile 来加载模块。否则，就会去找 C 程序库。

搜索的文件路径是从全局变量 package.cpath 获取，而这个变量则是通过环境变量 LUA_CPATH 来初始。

搜索的策略跟上面的一样，只不过现在换成搜索的是 so 或 dll 类型的文件。如果找得到，那么 require 就会通过 package.loadlib 来加载它。

C 包

Lua和C是很容易结合的，使用C为Lua写包。

与Lua中写包不同，C包在使用以前必须首先加载并连接，在大多数系统中最容易的实现方式是通过动态连接库机制。

Lua在一个叫loadlib的函数内提供了所有的动态连接的功能。这个函数有两个参数:库的绝对路径和初始化函数。所以典型的调用的例子如下:

```
local path = "/usr/local/lua/lib/libluasocket.so"
local f = loadlib(path, "luaopen_socket")
```

loadlib函数加载指定的库并且连接到Lua，然而它并不打开库（也就是说没有调用初始化函数），反之他返回初始化函数作为Lua的一个函数，这样我们就可以直接在Lua中调用他。

如果加载动态库或者查找初始化函数时出错，loadlib将返回nil和错误信息。我们可以修改前面一段代码，使其检测错误然后调用初始化函数：

```
local path = "/usr/local/lua/lib/libluasocket.so"
-- 或者 path = "C:\\windows\\luasocket.dll", 这是 Window 平台下
local f = assert(loadlib(path, "luaopen_socket"))
f()  -- 真正打开库
```

一般情况下我们期望二进制的发布库包含一个与前面代码段相似的stub文件，安装二进制库的时候可以随便放在某个目录，只需要修改stub文件对应二进制库的实际路径即可。

将stub文件所在的目录加入到LUA_PATH，这样设定后就可以使用require函数加载C库了。

Lua 元表(Metatable)

在 Lua table 中我们可以访问对应的key来得到value值，但是却无法对两个 table 进行操作。

因此 Lua 提供了元表(Metatable)，允许我们改变table的行为，每个行为关联了对应的元方法。

例如，使用元表我们可以定义Lua如何计算两个table的相加操作a+b。

当Lua试图对两个表进行相加时，先检查两者之一是否有元表，之后检查是否有一个叫"**add**"的字段，若找到，则调用对应的值。"add"等即时字段，其对应的值（往往是一个函数或是table）就是"元方法"。

有两个很重要的函数来处理元表：

- **setmetatable(table,metatable):** 对指定table设置元表(metatable)，如果元表(metatable)中存在__metatable键值，setmetatable会失败 。
- **getmetatable(table):** 返回对象的元表(metatable)。

以下实例演示了如何对指定的表设置元表：

```
mytable = {}
mymetatable = {}
setmetatable(mytable,mymetatable)
```

以上代码也可以直接写成一行：

```
mytable = setmetatable({},{})
```

__index 元方法

index 元方法查看表中元素是否存在，如果不存在，返回结果为**nil**；如果存在则由index 返回结果。

```
mytable = setmetatable({key1 = "value1"}, {
  __index = function(mytable, key)
    if key == "key2" then
      return "metatablevalue"
    else
      return mytable[key]
    end
  end
})

print(mytable.key1,mytable.key2)
```

实例输出结果为：

```
value1    metatablevalue
```

实例解析：

- mytable 表赋值为 **{key1 = "value1"}**。
- mytable 设置了元表，元方法为 __index。
- 在mytable表中查找 key1，如果找到，返回该元素，找不到则继续。
- 在mytable表中查找 key2，如果找到，返回该元素，找不到则继续。
- 判断元表有没有**index**方法，如果index方法是一个函数，则调用该函数。
- 元方法中查看是否传入 "key2" 键的参数（mytable.key2已设置），如果传入 "key2" 参数返回 "metatablevalue"，否则返回 mytable 对应的键值。

我们可以将以上代码简单写成：

```
mytable = setmetatable({key1 = "value1"}, { __index = { key2 = "metatablevalue" } })
print(mytable.key1,mytable.key2)
```

__newindex 元方法

newindex 元方法用来对表更新，index则用来对表访问 。

当你给表的一个缺少的索引赋值，解释器就会查找__newindex 元方法：如果存在则调用这个函数而不进行赋值操作。

以下实例演示了 __newindex 元方法的应用：

```
mymetatable = {}
mytable = setmetatable({key1 = "value1"}, { __newindex = mymetatable })

print(mytable.key1)

mytable.newkey = "新值2"
print(mytable.newkey,mymetatable.newkey)

mytable.key1 = "新值1"
print(mytable.key1,mymetatable.newkey1)
```

以上实例执行输出结果为：

```
value1
nil    新值2
新值1    nil
```

以上实例中表设置了元方法 **newindex**，在对新索引键（**newkey**）赋值时（**mytable.newkey = "新值2"**），会调用元方法，而不进行赋值。而如果对已存在的索引键（**key1**），则会进行赋值，而不调用元方法 newindex。

以下实例使用了 rawset 函数来更新表：

```
mytable = setmetatable({key1 = "value1"}, {
  __newindex = function(mytable, key, value)
        rawset(mytable, key, "\""..value.."\"")

  end
})

mytable.key1 = "new value"
mytable.key2 = 4

print(mytable.key1,mytable.key2)
```

以上实例执行输出结果为：

```
new value    "4"
```

为表添加操作符

以下实例演示了两表相加操作：

```
-- 计算表中最大值，table.maxn在Lua5.2以上版本中已无法使用
-- 自定义计算表中最大值函数 table_maxn
function table_maxn(t)
    local mn = 0
    for k, v in pairs(t) do
        if mn < k then
            mn = k
        end
    end
    return mn
end

-- 两表相加操作
mytable = setmetatable({ 1, 2, 3 }, {
  __add = function(mytable, newtable)
    for i = 1, table_maxn(newtable) do
      table.insert(mytable, table_maxn(mytable)+1,newtable[i])
    end
    return mytable
  end
})

secondtable = {4,5,6}

mytable = mytable + secondtable
    for k,v in ipairs(mytable) do
print(k,v)
end
```

以上实例执行输出结果为：

```
1    1
2    2
3    3
4    4
5    5
6    6
```

__add 键包含在元表中，并进行相加操作。 表中对应的操作列表如下：

模式	描述
__add	对应的运算符 '+'.
__sub	对应的运算符 '-'.
__mul	对应的运算符 '*'.
__div	对应的运算符 '/'.
__mod	对应的运算符 '%'.
__unm	对应的运算符 '-'.
__concat	对应的运算符 '..'.
__eq	对应的运算符 '=='.
__lt	对应的运算符 '<'.
__le	对应的运算符 '<='.

__call 元方法

__call 元方法在 Lua 调用一个值时调用。以下实例演示了计算表中元素的和：

```
-- 计算表中最大值，table.maxn在Lua5.2以上版本中已无法使用
-- 自定义计算表中最大值函数 table_maxn
function table_maxn(t)
    local mn = 0
    for k, v in pairs(t) do
        if mn < k then
            mn = k
        end
    end
    return mn
end

-- 定义元方法__call
mytable = setmetatable({10}, {
  __call = function(mytable, newtable)
    sum = 0
    for i = 1, table_maxn(mytable) do
        sum = sum + mytable[i]
    end
    for i = 1, table_maxn(newtable) do
        sum = sum + newtable[i]
    end
    return sum
  end
})
newtable = {10,20,30}
print(mytable(newtable))
```

以上实例执行输出结果为：

```
70
```

__tostring 元方法

__tostring 元方法用于修改表的输出行为。以下实例我们自定义了表的输出内容：

```
mytable = setmetatable({ 10, 20, 30 }, {
  __tostring = function(mytable)
    sum = 0
    for k, v in pairs(mytable) do
        sum = sum + v
    end
    return "表所有元素的和为 " .. sum
  end
})
print(mytable)
```

以上实例执行输出结果为：

```
表所有元素的和为 60
```

从本文中我们可以知道元表可以很好的简化我们的代码功能，所以了解 Lua 的元表，可以让我们写出更加简单优秀的 Lua 代码。

Lua 协同程序(coroutine)

什么是协同(coroutine)？

Lua 协同程序(coroutine)与线程比较类似：拥有独立的堆栈，独立的局部变量，独立的指令指针，同时又与其它协同程序共享全局变量和其它大部分东西。

协同是非常强大的功能，但是用起来也很复杂。

线程和协同程序区别

线程与协同程序的主要区别在于，一个具有多个线程的程序可以同时运行几个线程，而协同程序却需要彼此协作的运行。

在任一指定时刻只有一个协同程序在运行，并且这个正在运行的协同程序只有在明确的被要求挂起的时候才会被挂起。

协同程序有点类似同步的多线程，在等待同一个线程锁的几个线程有点类似协同。

基本语法

方法	描述
coroutine.create()	创建coroutine，返回coroutine， 参数是一个函数，当和resume配合使用的时候就唤醒函数调用
coroutine.resume()	重启coroutine，和create配合使用
coroutine.yield()	挂起coroutine，将coroutine设置为挂起状态，这个和resume配合使用能有很多有用的效果
coroutine.status()	查看coroutine的状态 注：coroutine的状态有三种：dead，suspend，running，具体什么时候有这样的状态请参考下面的程序
coroutine.wrap（）	创建coroutine，返回一个函数，一旦你调用这个函数，就进入coroutine，和create功能重复
coroutine.running()	返回正在跑的coroutine，一个coroutine就是一个线程，当使用running的时候，就是返回一个corouting的线程号

以下实例演示了以上各个方法的用法：

```
-- coroutine_test.lua 文件
co = coroutine.create(
    function(i)
        print(i);
    end
)

coroutine.resume(co, 1)   -- 1
print(coroutine.status(co))  -- dead

print("----------")

co = coroutine.wrap(
    function(i)
        print(i);
    end
)

co(1)

print("----------")

co2 = coroutine.create(
    function()
        for i=1,10 do
            print(i)
            if i == 3 then
                print(coroutine.status(co2))  --running
                print(coroutine.running()) --thread:XXXXXX
            end
            coroutine.yield()
        end
    end
)

coroutine.resume(co2) --1
coroutine.resume(co2) --2
coroutine.resume(co2) --3

print(coroutine.status(co2))   -- suspended
print(coroutine.running())   --nil

print("----------")
```

以上实例执行输出结果为：

```
1
dead
----------
1
----------
1
2
3
running
thread: 0x7fb801c05868    false
suspended
thread: 0x7fb801c04c88    true
----------
```

coroutine.running就可以看出来,coroutine在底层实现就是一个线程。

当create一个coroutine的时候就是在新线程中注册了一个事件。

当使用resume触发事件的时候，create的coroutine函数就被执行了，当遇到yield的时候就代表挂起当前线程，等候再次resume触发事件。

接下来我们分析一个更详细的实例：

```
function foo (a)
    print("foo 函数输出", a)
    return coroutine.yield(2 * a) -- 返回  2*a 的值
end

co = coroutine.create(function (a , b)
    print("第一次协同程序执行输出", a, b) -- co-body 1 10
    local r = foo(a + 1)

    print("第二次协同程序执行输出", r)
    local r, s = coroutine.yield(a + b, a - b)  -- a，b的值为第一次调用协同程序时传入

    print("第三次协同程序执行输出", r, s)
    return b, "结束协同程序"                   -- b的值为第二次调用协同程序时传入
end)

print("main", coroutine.resume(co, 1, 10)) -- true, 4
print("--分割线----")
print("main", coroutine.resume(co, "r")) -- true 11 -9
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- true 10 end
print("---分割线---")
print("main", coroutine.resume(co, "x", "y")) -- cannot resume dead coroutine
print("---分割线---")
```

以上实例执行输出结果为：

```
第一次协同程序执行输出    1    10
foo 函数输出    2
main    true    4
--分割线----
第二次协同程序执行输出    r
main    true    11    -9
---分割线---
第三次协同程序执行输出    x    y
main    true    10    结束协同程序
---分割线---
main    false    cannot resume dead coroutine
---分割线---
```

以上实例接下如下：

- 调用resume，将协同程序唤醒,resume操作成功返回true，否则返回false；
- 协同程序运行；
- 运行到yield语句；
- yield挂起协同程序，第一次resume返回；（注意：此处yield返回，参数是resume的参数）
- 第二次resume，再次唤醒协同程序；（注意：此处resume的参数中，除了第一个参数，剩下的参数将作为yield的参数）
- yield返回；
- 协同程序继续运行；

- 如果使用的协同程序继续运行完成后继续调用 resumev方法则输出：cannot resume dead coroutine

resume和yield的配合强大之处在于，resume处于主程中，它将外部状态（数据）传入到协同程序内部；而yield则将内部的状态（数据）返回到主程中。

生产者-消费者问题

现在我就使用Lua的协同程序来完成生产者-消费者这一经典问题。

```
local newProductor

function productor()
     local i = 0
     while true do
          i = i + 1
          send(i)     -- 将生产的物品发送给消费者
     end
end

function consumer()
     while true do
          local i = receive()     -- 从生产者那里得到物品
          print(i)
     end
end

function receive()
     local status, value = coroutine.resume(newProductor)
     return value
end

function send(x)
     coroutine.yield(x)     -- x表示需要发送的值，值返回以后，就挂起该协同程序
end

-- 启动程序
newProductor = coroutine.create(productor)
consumer()
```

```
local newProductor

function productor()
     local i = 0
     while true do
          i = i + 1
          send(i)     -- 将生产的物品发送给消费者
     end
end

function consumer()
     while true do
          local i = receive()     -- 从生产者那里得到物品
          print(i)
     end
end

function receive()
     local status, value = coroutine.resume(newProductor)
     return value
end

function send(x)
     coroutine.yield(x)     -- x表示需要发送的值，值返回以后，就挂起该协同程序
end

-- 启动程序
newProductor = coroutine.create(productor)
consumer()
```

以上实例执行输出结果为：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
……
```

Lua 文件 I/O

Lua I/O 库用于读取和处理文件。分为简单模式（和C一样）、完全模式。

- 简单模式（simple model）拥有一个当前输入文件和一个当前输出文件，并且提供针对这些文件相关的操作。
- 完全模式（complete model） 使用外部的文件句柄来实现。它以一种面对对象的形式，将所有的文件操作定义为文件句柄的方法

简单模式在做一些简单的文件操作时较为合适。但是在进行一些高级的文件操作的时候，简单模式就显得力不从心。例如同时读取多个文件这样的操作，使用完全模式则较为合适。

打开文件操作语句如下：

```
file = io.open (filename [, mode])
```

mode 的值有：

模式	描述
r	以只读方式打开文件，该文件必须存在。
w	打开只写文件，若文件存在则文件长度清为0，即该文件内容会消失。若文件不存在则建立该文件。
a	以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。（EOF符保留）
r+	以可读写方式打开文件，该文件必须存在。
w+	打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。
a+	与a类似，但此文件可读可写
b	二进制模式，如果文件是二进制文件，可以加上b
+	号表示对文件既可以读也可以写

简单模式

简单模式使用标准的 I/O 或使用一个当前输入文件和一个当前输出文件。

以下为 file.lua 文件代码，操作的文件为test.lua(如果没有你需要创建该文件)，代码如下：

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

-- 设置默认输入文件为 test.lua
io.input(file)

-- 输出文件第一行
print(io.read())

-- 关闭打开的文件
io.close(file)

-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")

-- 设置默认输出文件为 test.lua
io.output(file)

-- 在文件最后一行添加 Lua 注释
io.write("--  test.lua 文件末尾注释")

-- 关闭打开的文件
io.close(file)
```

执行以上代码，你会发现，输出了 test.ua 文件的第一行信息，并在该文件最后一行添加了 lua 的注释。如我这边输出的是：

```
-- test.lua 文件
```

在以上实例中我们使用了 io."x" 方法，其中 io.read() 中我们没有带参数，参数可以是下表中的一个：

模式	描述
"*n"	读取一个数字并返回它。例：file.read("*n")
"*a"	从当前位置读取整个文件。例：file.read("*a")
"*l"（默认）	读取下一行，在文件尾 (EOF) 处返回 nil。例：file.read("*l")
number	返回一个指定字符个数的字符串，或在 EOF 时返回 nil。例：file.read(5)

其他的 io 方法有：

- **io.tmpfile():**返回一个临时文件句柄，该文件以更新模式打开，程序结束时自动删除
- **io.type(file):** 检测obj是否一个可用的文件句柄
- **io.flush():** 向文件写入缓冲中的所有数据
- **io.lines(optional file name):** 返回一个迭代函数,每次调用将获得文件中的一行内容,当到文件尾时，将返回nil,但不关闭文件

完全模式

通常我们需要在同一时间处理多个文件。我们需要使用 file:function_name 来代替 io.function_name 方法。以下实例演示了如同同时处理同一个文件:

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

-- 输出文件第一行
print(file:read())

-- 关闭打开的文件
file:close()

-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")

-- 在文件最后一行添加 Lua 注释
file:write("--test")

-- 关闭打开的文件
file:close()
```

执行以上代码，你会发现，输出了 test.ua 文件的第一行信息，并在该文件最后一行添加了 lua 的注释。如我这边输出的是：

```
-- test.lua 文件
```

read 的参数与简单模式一致。

其他方法:

- **file:seek(optional whence, optional offset):** 设置和获取当前文件位置,成功则返回最终的文件位置(按字节),失败则返回nil加错误信息。参数 whence 值可以是:
  - "set": 从文件头开始
  - "cur": 从当前位置开始[默认]
  - "end": 从文件尾开始
  - offset:默认为0不带参数file:seek()则返回当前位置,file:seek("set")则定位到文件头,file:seek("end")则定位到文件尾并返回文件大小
- **file:flush():** 向文件写入缓冲中的所有数据
- **io.lines(optional file name):** 打开指定的文件filename为读模式并返回一个迭代函数,每次调用将获得文件中的一行内容,当到文件尾时，将返回nil,并自动关闭文件。 若不带参数时io.lines() <=> io.input():lines(); 读取默认输入设备的内容，但结束时不关闭文件,如

  ```
  for line in io.lines("main.lua") do

      print(line)

      end
  ```

以下实例使用了 seek 方法，定位到文件倒数第 25 个位置并使用 read 方法的 *a 参数，即从当期位置(倒数第 25 个位置)读取整个文件。

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

file:seek("end",-25)
print(file:read("*a"))

-- 关闭打开的文件
file:close()
```

我这边输出的结果是：

```
st.lua 文件末尾--test
```

Lua 错误处理

程序运行中错误处理是必要的，在我们进行文件操作，数据转移及web service 调用过程中都会出现不可预期的错误。如果不注重错误信息的处理，就会照成信息泄露，程序无法运行等情况。

任何程序语言中，都需要错误处理。错误类型有：

- 语法错误
- 运行错误

语法错误

语法错误通常是由于对程序的组件（如运算符、表达式）使用不当引起的。一个简单的实例如下：

```
-- test.lua 文件
a == 2
```

以上代码执行结果为：

```
lua: test.lua:2: syntax error near '=='
```

正如你所看到的，以上出现了语法错误，一个 "=" 号跟两个 "=" 号是有区别的。一个 "=" 是赋值表达式两个 "=" 是比较运算。

另外一个实例:

```
for a= 1,10
   print(a)
end
```

执行以上程序会出现如下错误：

```
lua: test2.lua:2: 'do' expected near 'print'
```

语法错误比程序运行错误更简单，运行错误无法定位具体错误，而语法错误我们可以很快的解决，如以上实例我们只要在for语句下添加 do 即可：

```
for a= 1,10
do
   print(a)
end
```

运行错误

运行错误是程序可以正常执行，但是会输出报错信息。如下实例由于参数输入错误，程序执行时报错：

```
function add(a,b)
   return a+b
end

add(10)
```

当我们编译运行以下代码时，编译是可以成功的，但在运行的时候会产生如下错误：

```
lua: test2.lua:2: attempt to perform arithmetic on local 'b' (a nil value)
stack traceback:
    test2.lua:2: in function 'add'
    test2.lua:5: in main chunk
    [C]: ?
```

以下报错信息是由于程序缺少 b 参数引起的。

错误处理

我们可以使用两个函数：assert 和 error 来处理错误。实例如下：

```
local function add(a,b)
   assert(type(a) == "number", "a 不是一个数字")
   assert(type(b) == "number", "b 不是一个数字")
   return a+b
end
add(10)
```

执行以上程序会出现如下错误：

```
lua: test.lua:3: b 不是一个数字
stack traceback:
    [C]: in function 'assert'
    test.lua:3: in local 'add'
    test.lua:6: in main chunk
    [C]: in ?
```

实例中assert首先检查第一个参数，若没问题，assert不做任何事情；否则，assert以第二个参数作为错误信息抛出。

error函数

语法格式：

```
error (message [, level])
```

功能：终止正在执行的函数，并返回message的内容作为错误信息(error函数永远都不会返回)

通常情况下，error会附加一些错误位置的信息到message头部。

Level参数指示获得错误的位置:

- Level=1[默认]：为调用error位置(文件+行号)
- Level=2：指出哪个调用error的函数的函数
- Level=0:不添加错误位置信息

pcall 和 xpcall、debug

Lua中处理错误，可以使用函数pcall（protected call）来包装需要执行的代码。

pcall接收一个函数和要传递个后者的参数，并执行，执行结果：有错误、无错误；返回值true或者或false, errorinfo。

语法格式如下

```
if pcall(function_name, ….) then
-- 没有错误
else
-- 一些错误
end
```

简单实例：

```
> =pcall(function(i) print(i) end, 33)
33
true

> =pcall(function(i) print(i) error('error..') end, 33)
33
false        stdin:1: error..
```

```
> function f() return false,2 end
> if f() then print '1' else print '0' end
0
```

pcall以一种"保护模式"来调用第一个参数，因此pcall可以捕获函数执行中的任何错误。

通常在错误发生时，希望落得更多的调试信息，而不只是发生错误的位置。但pcall返回时，它已经销毁了调用栈的部分内容。

Lua提供了xpcall函数，xpcall接收第二个参数——一个错误处理函数，当错误发生时，Lua会在调用栈展看（unwind）前调用错误处理函数，于是就可以在这个函数中使用debug库来获取关于错误的额外信息了。

debug库提供了两个通用的错误处理函数:

- debug.debug：提供一个Lua提示符，让用户来价差错误的原因
- debug.traceback：根据调用栈来构建一个扩展的错误消息

> =xpcall(function(i) print(i) error('error..') end, function() print(debug.traceback()) end, 33) 33 stack traceback: stdin:1: in function 1>[C]: in function 'error' stdin:1: in function 1>[C]: in function 'xpcall' stdin:1: in main chunk [C]: in ? false nil
>
> 1>

1>

xpcall 使用实例 2:

```
function myfunction ()
   n = n/nil
end

function myerrorhandler( err )
   print( "ERROR:", err )
end

status = xpcall( myfunction, myerrorhandler )
print( status)
```

执行以上程序会出现如下错误：

```
ERROR:    test2.lua:2: attempt to perform arithmetic on global 'n' (a nil value)
false
```

Lua 调试(Debug)

Lua 提供了 debug 库用于提供创建我们自定义调速器的功能。Lua 本身并未有内置的调速器，但很多开发者共享了他们的 Lua 调速器代码。

Lua 中 debug 库包含以下函数：

sethook ([thread,] hook, mask [, count]):

方法	用途
debug():	进入一个用户交互模式，运行用户输入的每个字符串。 使用简单的命令以及其它调试设置，用户可以检阅全局变量和局部变量， 改变变量的值，计算一些表达式，等等。输入一行仅包含 cont 的字符串将结束这个函数， 这样调用者就可以继续向下运行。
getfenv(object):	返回对象的环境变量。
gethook(optional thread):	返回三个表示线程钩子设置的值： 当前钩子函数，当前钩子掩码，当前钩子计数
getinfo ([thread,] f [, what]):	返回关于一个函数信息的表。 你可以直接提供该函数， 也可以用一个数字 f 表示该函数。 数字 f 表示运行在指定线程的调用栈对应层次上的函数： 0 层表示当前函数（getinfo 自身）； 1 层表示调用 getinfo 的函数 （除非是尾调用，这种情况不计入栈）；等等。 如果 f 是一个比活动函数数量还大的数字， getinfo 返回 nil。
debug.getlocal ([thread,] f, local):	此函数返回在栈的 f 层处函数的索引为 local 的局部变量 的名字和值。 这个函数不仅用于访问显式定义的局部变量，也包括形参、临时变量等。
getmetatable(value):	把给定索引指向的值的元表压入堆栈。如果索引无效，或是这个值没有元表，函数将返回 0 并且不会向栈上压任何东西。
getregistry():	返回注册表表，这是一个预定义出来的表， 可以用来保存任何 C 代码想保存的 Lua 值。
getupvalue (f, up)	此函数返回函数 f 的第 up 个上值的名字和值。 如果该函数没有那个上值，返回 nil 。以 '(' （开括号）打头的变量名表示没有名字的变量 （去除了调试信息的代码块）。
	将一个函数作为钩子函数设入。 字符串 mask 以及数字 count 决定了钩子将在何时调用。 掩码是由下列字符组合成的字符串，每个字符有其含义： '`c`': 每当 Lua 调用一个函数时，调用钩子； '`r`': 每当 Lua 从一个函数内返回时，调用钩子； '`l`': 每当 Lua 进入新的一行时，调用钩子。
setlocal ([thread,] level, local, value):	这个函数将 value 赋给 栈上第 level 层函数的第 local 个局部变量。 如果没有那个变量，函数返回 nil 。 如果 level 越界，抛出一个错误。
setmetatable (value, table):	将 value 的元表设为 table （可以是 nil）。 返回 value。
setupvalue (f, up, value):	这个函数将 value 设为函数 f 的第 up 个上值。 如果函数没有那个上值，返回 nil 否则，返回该上值的名字。
traceback ([thread,] [message [, level]]):	如果 message 有，且不是字符串或 nil， 函数不做任何处理直接返回 message。 否则，它返回调用栈的栈回溯信息。 字符串可选项 message 被添加在栈回溯信息的开头。 数字可选项 level 指明从栈的哪一层开始回溯 （默认为 1 ，即调用 traceback 的那里）。

上表列出了我们常用的调试函数，接下来我们可以看些简单的例子：

```
function myfunction ()
print(debug.traceback("Stack trace"))
print(debug.getinfo(1))
print("Stack trace end")
    return 10
end
myfunction ()
print(debug.getinfo(1))
```

执行以上代码输出结果为：

```
Stack trace
stack traceback:
    test2.lua:2: in function 'myfunction'
    test2.lua:8: in main chunk
    [C]: ?
table: 0054C6C8
Stack trace end
```

在以实例中，我们使用到了 debug 库的 traceback 和 getinfo 函数， getinfo 函数用于返回函数信息的表。

另一个实例

我们经常需要调试函数的内的局部变量。我们可以使用 getupvalue 函数来设置这些局部变量。实例如下：

```
function newCounter ()
  local n = 0
  local k = 0
  return function ()
    k = n
    n = n + 1
    return n
    end
end

counter = newCounter ()
print(counter())
print(counter())

local i = 1

repeat
  name, val = debug.getupvalue(counter, i)
  if name then
    print ("index", i, name, "=", val)
    if(name == "n") then
        debug.setupvalue (counter,2,10)
    end
    i = i + 1
  end -- if
until not name

print(counter())
```

执行以上代码输出结果为：

```
1
2
index    1    k    =    1
index    2    n    =    2
11
```

在以上实例中，计数器在每次调用时都会自增1。实例中我们使用了 getupvalue 函数查看局部变量的当前状态。我们可以设置局部变量为新值。实例中，在设置前 n 的值为 2,使用 setupvalue 函数将其设置为 10。现在我们调用函数，执行后输出为 11 而不是 3。

调试类型

- 命令行调试
- 图形界面调试

命令行调试器有：RemDebug、clidebugger、ctrace、xdbLua、LuaInterface - Debugger、Rldb、ModDebug。

图形界调试器有：SciTE、Decoda、ZeroBrane Studio、akdebugger、luaedit。

Lua 垃圾回收

Lua 采用了自动内存管理。 这意味着你不用操心新创建的对象需要的内存如何分配出来， 也不用考虑在对象不再被使用后怎样释放它们所占用的内存。

Lua 运行了一个垃圾收集器来收集所有死对象 （即在 Lua 中不可能再访问到的对象）来完成自动内存管理的工作。 Lua 中所有用到的内存，如：字符串、表、用户数据、函数、线程、内部结构等，都服从自动管理。

Lua 实现了一个增量标记-扫描收集器。 它使用这两个数字来控制垃圾收集循环： 垃圾收集器间歇率和垃圾收集器步进倍率。 这两个数字都使用百分数为单位 （例如：值 100 在内部表示 1 ）。

垃圾收集器间歇率控制着收集器需要在开启新的循环前要等待多久。 增大这个值会减少收集器的积极性。 当这个值比 100 小的时候，收集器在开启新的循环前不会有等待。 设置这个值为 200 就会让收集器等到总内存使用量达到 之前的两倍时才开始新的循环。

垃圾收集器步进倍率控制着收集器运作速度相对于内存分配速度的倍率。 增大这个值不仅会让收集器更加积极，还会增加每个增量步骤的长度。 不要把这个值设得小于 100 ， 那样的话收集器就工作的太慢了以至于永远都干不完一个循环。 默认值是 200 ， 这表示收集器以内存分配的"两倍"速工作。

如果你把步进倍率设为一个非常大的数字 （比你的程序可能用到的字节数还大 10% ）， 收集器的行为就像一个 stop-the-world 收集器。 接着你若把间歇率设为 200 ， 收集器的行为就和过去的 Lua 版本一样了： 每次 Lua 使用的内存翻倍时，就做一次完整的收集。

垃圾回收器函数

Lua 提供了以下函数**collectgarbage ([opt [, arg]])**用来控制自动内存管理:

- **collectgarbage("collect"):** 做一次完整的垃圾收集循环。通过参数 opt 它提供了一组不同的功能：
- **collectgarbage("count"):** 以 K 字节数为单位返回 Lua 使用的总内存数。 这个值有小数部分，所以只需要乘上 1024 就能得到 Lua 使用的准确字节数（除非溢出）。
- **collectgarbage("restart"):** 重启垃圾收集器的自动运行。
- **collectgarbage("setpause"):** 将 arg 设为收集器的 间歇率 （参见 §2.5）。 返回 间歇率 的前一个值。
- **collectgarbage("setstepmul"):** 返回 步进倍率 的前一个值。

- **collectgarbage("step"):** 单步运行垃圾收集器。 步长"大小"由 arg 控制。 传入 0 时，收集器步进（不可分割的）一步。 传入非 0 值， 收集器收集相当于 Lua 分配这些多（K 字节）内存的工作。 如果收集器结束一个循环将返回 true 。
- **collectgarbage("stop"):** 停止垃圾收集器的运行。 在调用重启前，收集器只会因显式的调用运行。

以下演示了一个简单的垃圾回收实例:

```
mytable = {"apple", "orange", "banana"}

print(collectgarbage("count"))

mytable = nil

print(collectgarbage("count"))

print(collectgarbage("collect"))

print(collectgarbage("count"))
```

执行以上程序，输出结果如下(注意内存使用的变化)：

```
20.9560546875
20.9853515625
0
19.4111328125
```

Lua 面向对象

面向对象编程（Object Oriented Programming，OOP）是一种非常流行的计算机编程架构。

以下几种编程语言都支持面向对象编程：

- C++
- Java
- Objective-C
- Smalltalk
- C#
- Ruby

面向对象特征

- 1） 封装：指能够把一个实体的信息、功能、响应都装入一个单独的对象中的特性。
- 2） 继承：继承的方法允许在不改动原程序的基础上对其进行扩充，这样使得原功能得以保存，而新功能也得以扩展。这有利于减少重复编码，提高软件的开发效率。
- 3） 多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。在运行时，可以通过指向基类的指针，来调用实现派生类中的方法。
- 4）抽象：抽象(Abstraction)是简化复杂的现实问题的途径，它可以为具体问题找到最恰当的类定义，并且可以在最恰当的继承级别解释问题。

Lua 中面向对象

我们知道，对象由属性和方法组成。LUA中最基本的结构是table，所以需要用table来描述对象的属性。

lua中的function可以用来表示方法。那么LUA中的类可以通过table + function模拟出来。

至于继承，可以通过metetable模拟出来（不推荐用，只模拟最基本的对象大部分时间够用了）。

Lua中的表不仅在某种意义上是一种对象。像对象一样，表也有状态（成员变量）；也有与对象的值独立的本性，特别是拥有两个不同值的对象（table）代表两个不同的对象；一个对象在不同的时候也可以有不同的值，但他始终是一个对象；与对象类似，表的生命周期与其由什么创建、在哪创建没有关系。对象有他们的成员函数，表也有：

```
Account = {balance = 0}
function Account.withdraw (v)
    Account.balance = Account.balance - v
end
```

这个定义创建了一个新的函数，并且保存在Account对象的withdraw域内，下面我们可以这样调用：

```
Account.withdraw(100.00)
```

一个简单实例

以下简单的类包含了三个属性： area, length 和 breadth，printArea方法用于打印计算结果：

```
-- Meta class
Rectangle = {area = 0, length = 0, breadth = 0}

-- 派生类的方法 new
function Rectangle:new (o,length,breadth)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  self.length = length or 0
  self.breadth = breadth or 0
  self.area = length*breadth;
  return o
end

-- 派生类的方法 printArea
function Rectangle:printArea ()
  print("矩形面积为 ",self.area)
end
```

创建对象

创建对象是位类的实例分配内存的过程。每个类都有属于自己的内存并共享公共数据。

```
r = Rectangle:new(nil,10,20)
```

访问属性

我们可以使用点号(.)来访问类的属性：

```
print(r.length)
```

访问成员函数

我们可以使用冒号(:)来访问类的属性：

```
r:printArea()
```

内存在对象初始化时分配。

完整实例

以下我们演示了 Lua 面向对象的完整实例：

```
-- Meta class
Shape = {area = 0}

-- 基础类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  side = side or 0
  self.area = side*side;
  return o
end

-- 基础类方法 printArea
function Shape:printArea ()
  print("面积为 ",self.area)
end

-- 创建对象
myshape = Shape:new(nil,10)

myshape:printArea()
```

执行以上程序，输出结果为：

```
面积为    100
```

Lua 继承

继承是指一个对象直接使用另一对象的属性和方法。可用于扩展基础类的属性和方法。

以下演示了一个简单的继承实例：

```
 -- Meta class
Shape = {area = 0}
-- 基础类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  side = side or 0
  self.area = side*side;
  return o
end
-- 基础类方法 printArea
function Shape:printArea ()
  print("面积为 ",self.area)
end
```

接下来的实例，Square 对象继承了 Shape 类:

```
Square = Shape:new()
-- Derived class method new
function Square:new (o,side)
  o = o or Shape:new(o,side)
  setmetatable(o, self)
  self.__index = self
  return o
end
```

完整实例

以下实例我们继承了一个简单的类，来扩展派生类的方法，派生类中保留了继承类的成员变量和方法：

```
 -- Meta class
Shape = {area = 0}
-- 基础类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  side = side or 0
  self.area = side*side;
  return o
end
-- 基础类方法 printArea
function Shape:printArea ()
  print("面积为 ",self.area)
end

-- 创建对象
myshape = Shape:new(nil,10)
myshape:printArea()

Square = Shape:new()
-- 派生类方法 new
function Square:new (o,side)
  o = o or Shape:new(o,side)
  setmetatable(o, self)
  self.__index = self
  return o
end

-- 派生类方法 printArea
function Square:printArea ()
  print("正方形面积为 ",self.area)
end

-- 创建对象
mysquare = Square:new(nil,10)
mysquare:printArea()

Rectangle = Shape:new()
-- 派生类方法 new
function Rectangle:new (o,length,breadth)
  o = o or Shape:new(o)
  setmetatable(o, self)
  self.__index = self
  self.area = length * breadth
  return o
end

-- 派生类方法 printArea
function Rectangle:printArea ()
  print("矩形面积为 ",self.area)
end

-- 创建对象
myrectangle = Rectangle:new(nil,10,20)
myrectangle:printArea()
```

执行以上代码，输出结果为：

```
面积为 	100
正方形面积为 	100
矩形面积为 	200
```

函数重写

Lua 中我们可以重写基础类的函数，在派生类中定义自己的实现方式：

```
-- 派生类方法 printArea
function Square:printArea ()
  print("正方形面积 ",self.area)
end
```

Lua 数据库访问

本文主要为大家介绍 Lua 数据库的操作库：LuaSQL。他是开源的，支持的数据库有：ODBC, ADO, Oracle, MySQL, SQLite 和 PostgreSQL。

本文为大家介绍MySQL的数据库连接。

LuaSQL 可以使用 LuaRocks 来安装可以根据需要安装你需要的数据库驱动。

LuaRocks 安装方法：

```
$ wget http://luarocks.org/releases/luarocks-2.2.1.tar.gz
$ tar zxpf luarocks-2.2.1.tar.gz
$ cd luarocks-2.2.1
$ ./configure; sudo make bootstrap
$ sudo luarocks install luasocket
$ lua
Lua 5.3.0 Copyright (C) 1994-2015 Lua.org, PUC-Rio
> require "socket"
```

Window 下安装 LuaRocks：https://github.com/keplerproject/luarocks/wiki/Installation-instructions-for-Windows

安装不同数据库驱动：

```
luarocks install luasql-sqlite3
luarocks install luasql-postgres
luarocks install luasql-mysql
luarocks install luasql-sqlite
luarocks install luasql-odbc
```

你也可以使用源码安装方式，Lua Github 源码地址：https://github.com/keplerproject/luasql

Lua 连接MySql 数据库：

```
require "luasql.mysql"

--创建环境对象
env = luasql.mysql()

--连接数据库
conn = env:connect("数据库名","用户名","密码","IP地址",端口)

--设置数据库的编码格式
conn:execute"SET NAMES UTF8"

--执行数据库操作
cur = conn:execute("select * from role")

row = cur:fetch({},"a")

--文件对象的创建
file = io.open("role.txt","w+");

while row do
    var = string.format("%d %s\n", row.id, row.name)

    print(var)

    file:write(var)

    row = cur:fetch(row,"a")
end

file:close()  --关闭文件对象
conn:close()  --关闭数据库连接
env:close()   --关闭数据库环境
```

</stdin:1></stdin:1></p这里注意对返回值的逻辑判断：<>